IN-2194 Warm-Up Exercise: GNUnet

1 GNUnet: Getting Started

First of all you have to install a current version of GNUnet. You can obtain the latest development version from subversion by cloning the repository using:

```
svn checkout https://gnunet.org/svn/gnunet
```

or, once released, you will be able to download a tarball of a more stable version from our website:

`ftp://ftp.gnu.org/gnu/gnunet/gnunet-0.9.0pre3.tar.gz`

Detailed installation instructions for various operating systems and a detailed list of all dependencies can found on our website:
`https://gnunet.org/installation`

```
configure --prefix=$HOME
make
make install
export GNUNET_PREFIX=$HOME
export PATH=$PATH:$GNUNET_PREFIX/bin
make check
mkdir .gnunet/
touch .gnunet/gnunet.conf
```

The configure prefix defines where to install GNUnet. If you do not specifiy a prefix, GNUnet is installed your home directory. For development purposes you can specify a directory like `/tmp/gnunet`.

You should add the **export** GNUNET_PREFIX=$HOME to your `.bash_rc` or `.profile` to be sure it is always set.

1.1 Baby Steps

First, you may want to just generate the peer's private key:

```
gnunet-peerinfo -s
\end{litlisting}

GNUnet can then be started with \lstinline|gnunet-arm -s| and stopped with
\lstinline|gnunet-arm -e|. You can specify a configuration for all
services using the $-c$ switch.

\begin{lstlisting}
gnunet-arm -i dht
gnunet-dht-put -k KEY -d VALUE
gnunet-dht-get -k KEY
gnunet-statistics
gnunet-statistics -s dht
```

1.2 Starting Two Peers by Hand

For the second peer, you will need to manually create a modified configuration file to avoid conflicts with ports and directories. Use `$GNUNET_PREFIX/share/gnunet/defaults.conf` as a template (create a copy called `peer2.conf`) and change:

- SERVICEHOME under PATHS
- Every value for “PORT” (add 10000) in any section
- Every value for “UNIXPATH” in any section

Now, generate the 2nd peer’s private key:

```
gnunet-peerinfo -s -c peer2.conf
\end{litlisting}

This may take a while, generate entropy using your keyboard or mouse
as needed.  Also, make sure the output is different from the {\tt
  gnunet-peerinfo} output for the first peer (otherwise you made an
error in the configuration).

Then, you can start a second peer using:
\begin{lstlisting}
gnunet-arm -c peer2.conf -s
gnunet-arm -c peer2.conf -i dht
gnunet-dht-put -c peer2.conf -k KEY -d VALUE
gnunet-dht-get -c peer2.conf -k KEY
```

If you want the two peers to connect, change the first one to be a hostlist server by:

- Creating an empty configuration file `peer1.conf`
- Adding a line in the “[hostlist]” section with “OPTIONS = -p”

Then change `peer2.conf`, replacing the “SERVERS” line with “http://localhost:8080/”. Restart both peers using `gnunet-arm`. Check that they are connected using `gnunet-statistics -s core`.

2 gnunet-ext

A template build system for writing GNUnet extensions in C can be obtained as follows:

```
svn checkout https://gnunet.org/svn/gnunet-ext/
cd gnunet-ext/
. bootstrap
./configure --prefix=$HOME --with-gnunet=$GNUNET_PREFIX
make
make install
make check
```

The first step for writing any extension with a new service is to ensure that the `defaults.conf` file contains entries for the UNIXPATH, PORT and BINARYNAME for the service in a section named after the service. The `defaults.conf` is located in the `$GNUNET_PREFIX/share/gnunet/` directory (and will be overwritten if you run `make install` for the main GNUnet distribution afterwards, so be careful about this!).[1]

3 GNUnet Architecture

GNUnet is organized in layers or services. Each service is composed of a main service implementation and a client library for other programs to use, described by an API.

Very often it is other GNUnet services that will use these APIs to build the higher layers of GNUnet on top of the lower ones. Each layer expands or extends the functionality of the service below (for instance, to build a mesh on top of a DHT).

The main service implementation runs as a standalone process in the operating system and the client code runs as part of the client program, so crashes of a client do not affect the service process or other clients. The service and the clients communicate via a message protocol to be defined and implemented by the programmer.

4 Writing a Client

4.1 Writing a Client Application

When writing any client application (for example, a command-line tool), the basic structure is to start with the `GNUNET_PROGRAM_run` function. This function will parse command-line options, setup the scheduler and then invokes the `run` function (with the remaining non-option arguments) and a handle to the parsed configuration (and the configuration file name that was used, which is typically not needed):

```
#include <gnunet/gnunet_util_lib.h>

static void
run (void *cls,
     char *const *args,
     const char *cfgfile,
     const struct GNUNET_CONFIGURATION_Handle *cfg)
{
  /* main code here */
}

int
main (int argc, char *const *argv)
{
  static const struct GNUNET_GETOPT_CommandLineOption options[] = {
    GNUNET_GETOPT_OPTION_END
  };
  return (GNUNET_OK ==
          GNUNET_PROGRAM_run (argc,
                              argv,
```

[1]A more elegant solution for extending `defaults.conf` will be provided in the future.

```
                                        "binary-name",
                                        gettext_noop ("binary description text"),
                                        options, &run, NULL)) ? ret : 1;
}
```

Options can then be added easily by adding global variables and expanding the `options` array. For example, the following would add a string-option and a binary flag (defaulting to `NULL` and `GNUNET_NO` respectively):

```
static char *string_option;
static int a_flag;

// ...
  static const struct GNUNET_GETOPT_CommandLineOption options[] = {
  {'s', "name", "SOMESTRING",
     gettext_noop ("text describing the string_option NAME"), 1,
     &GNUNET_GETOPT_set_string, &string_option},
    {'f', "flag", NULL,
     gettext_noop ("text describing the flag option"), 0,
     &GNUNET_GETOPT_set_one, &a_flag},
    GNUNET_GETOPT_OPTION_END
  };
// ...
```

Issues such as displaying some helpful text describing options using the `--help` argument and error handling are taken care of when using this approach. Other `GNUNET_GETOPT_`-functions can be used to obtain integer value options, increment counters, etc. You can even write custom option parsers for special circumstances not covered by the available handlers.

Inside the `run` method, the program would perform the application-specific logic, which typically involves initializing and using some client library to interact with the service. The client library is supposed to implement the IPC whereas the service provides more persistent P2P functions.

4.2 Writing a Client Library

The first and most important step in writing a client library is to decide on an API for the library. Typical API calls include connecting to the service, performing application-specific requests and cleaning up. Many examples for such service APIs can be found in the `gnunet/src/include/gnunet_*_service.h` files.

Then, a client-service protocol needs to be designed. This typically involves defining various message formats in a header that will be included by both the service and the client library (but is otherwise not shared and hence located within the service's directory and not installed by `make install`). Each message must start with a `struct GNUNET_MessageHeader` and must be shorter than 64k. By convention, all fields in IPC (and P2P) messages must be in big-endian format (and thus should be read using `ntohl` and similar functions and written using `htonl` and similar functions). Unique message types must be defined for each message struct in the `gnunet_protocols.h` header (or an extension-specific include file).

4.3 Connecting to the Service

Before a client library can implement the application-specific protocol with the service, a connection must be created:

```
  client = GNUNET_CLIENT_connect ("service-name", cfg);
```

As a result a `GNUNET_CLIENT_Connection` handle is returned which has to used in later API calls related to this service. The complete client API can be found in `gnunet_client_lib.h`

4.4 GNUnet Messages

In GNUnet, messages are always sent beginning with a `struct GNUNET_MessageHeader` in big endian format. This header defines the size and the type of the message, the payload follows after this header.

```
struct GNUNET_MessageHeader
{

  /**
   * The length of the struct (in bytes, including the length field itself),
   * in big-endian format.
   */
  uint16_t size GNUNET_PACKED;

  /**
   * The type of the message (GNUNET_MESSAGE_TYPE_XXXX), in big-endian format.
   */
  uint16_t type GNUNET_PACKED;

};
```

Existing message types are defined in `gnunet_protocols.h`
A common way to create a message is:

```
struct GNUNET_MessageHeader *msg =
  GNUNET_malloc(payload_size + sizeof(struct GNUNET_MessageHeader));
msg->size = htons(payload_size + sizeof(struct GNUNET_MessageHeader));
msg->type = htons(GNUNET_MY_MESSAGE_TYPE);
memcpy(&msg[1], &payload, payload_size);
```

4.5 Sending Requests to the Service

Any client-service protocol must start with the client sending the first message to the service, since services are only notified about (new) clients upon receiving a the first message.

Clients can transmit messages to the service using the `GNUNET_CLIENT_notify_transmit_ready` API:

```
static size_t
transmit_cb (void *cls, size_t size, void *buf)
{
  // ...
  if (NULL == buf) { handle_error(); return 0; }
  GNUNET_assert (size >= msg_size);
  memcpy (buf, my_msg, msg_size);
  // ...
  return msg_size;
}

// ...
```

```
th = GNUNET_CLIENT_notify_transmit_ready (client,
                                           msg_size,
                                     timeout,
                                           GNUNET_YES,
                                           &transmit_cb, cls);
// ...
```

The client-service protocoll calls `GNUNET_CLIENT_notify_transmit_ready` to be notified when the client is ready to send data to the service. Besides other arguments, you have to pass the client returned from the `connect` call, the message size and the callback function to call when the client is ready to send.

Only a single transmission request can be queued per client at the same time using this API. The handle `th` can be used to cancel the request if necessary (for example, during shutdown).

When `transmit_cb` is called the message is copied in the buffer provided and the number of bytes copied into the buffer is returned. `transmit_cb` could also return 0 if for some reason no message could be constructed; this is not an error and the connection to the service will persist in this case.

4.6 Receiving Replies from the Service

Clients can receive messages from the service using the `GNUNET_CLIENT_receive` API:

```
/**
 * Function called with messages from stats service.
 *
 * @param cls closure
 * @param msg message received, NULL on timeout or fatal error
 */
static void
receive_message (void *cls, const struct GNUNET_MessageHeader *msg)
{
  struct MyArg *arg = cls;

  // process 'msg'
}

// ...
  GNUNET_CLIENT_receive (client,
                         &receive_message,
                         arg,
                         timeout);
// ...
```

It should be noted that this receive call only receives a single message. To receive additional messages, `GNUNET_CLIENT_receive` must be called again.

5 Writing a Service

5.1 Code Placement

New services are placed in their own subdirectory under `gnunet/src`. This subdirectory should contain the API implementation file `SERVICE_api.c`, the description of the client-service protocol `SERVICE.h` and P2P protocol `SERVICE_protocol.h`, the implementation of the service itself `gnunet-service-SERVICE.h` and several files for tests, including test code and configuration files.

5.2 Starting a Service

The key API definitions for starting services are"

```
typedef void (*GNUNET_SERVICE_Main) (void *cls,
                                     struct GNUNET_SERVER_Handle *server,
                                     const struct GNUNET_CONFIGURATION_Handle *cfg);
int GNUNET_SERVICE_run (int argc,
                        char *const *argv,
                        const char *serviceName,
                        enum GNUNET_SERVICE_Options opt,
                        GNUNET_SERVICE_Main task,
                        void *task_cls);
```

Here is a starting point for your main function for your service:

```
static void my_main (void *cls,
                     struct GNUNET_SERVER_Handle *server,
                     const struct GNUNET_CONFIGURATION_Handle *cfg)
{
  /* do work */
}

int main (int argc, char *const*argv)
{
  if (GNUNET_OK !=
      GNUNET_SERVICE_run (argc, argv, "my",
                          GNUNET_SERVICE_OPTION_NONE,
                          &my_main, NULL);
    return 1;
  return 0;
}
```

5.3 Receiving Requests from Clients

Inside of the `my_main` method, a service typically registers for the various message types from clients that it supports by providing a handler function, the message type itself and possibly a fixed message size (or 0 for variable-size messages):

```
static void
handle_set (void *cls,
             struct GNUNET_SERVER_Client *client,
             const struct GNUNET_MessageHeader *message)
{
  GNUNET_SERVER_receive_done (client, GNUNET_OK);
}
static void
handle_get (void *cls,
             struct GNUNET_SERVER_Client *client,
             const struct GNUNET_MessageHeader *message)
{
  GNUNET_SERVER_receive_done (client, GNUNET_OK);
}

static void my_main (void *cls,
                      struct GNUNET_SERVER_Handle *server,
                      const struct GNUNET_CONFIGURATION_Handle *cfg)
{
  static const struct GNUNET_SERVER_MessageHandler handlers[] = {
    {&handle_set, NULL, GNUNET_MESSAGE_TYPE_MYNAME_SET, 0},
    {&handle_get, NULL, GNUNET_MESSAGE_TYPE_MYNAME_GET, 0},
    {NULL, NULL, 0, 0}
  };
  GNUNET_SERVER_add_handlers (server, handlers);
   /* do more setup work */
}
```

Each handler function **must** eventually (possibly in some asynchronous continuation) call `GNUNET_SERVER_receive_done`. Only after this call additional messages from the same client may be processed. This way, the service can throttle processing messages from the same client. By passing `GNUNET_SYSERR`, the service can close the connection to the client, indicating an error.

Services must check that client requests are well-formed and must not crash on protocol violations by the clients. Similarly, client libraries must check replies from servers and should gracefully report errors via their API.

5.4 Responding to Clients

Servers can send messages to clients using the `GNUNET_SERVER_notify_transmit_ready` API:

```
static size_t
transmit_cb (void *cls, size_t size, void *buf)
{
  // ...
  if (NULL == buf) { handle_error(); return 0; }
  GNUNET_assert (size >= msg_size);
  memcpy (buf, my_msg, msg_size);
  // ...
  return msg_size;
}
```

```
// ...
th = GNUNET_SERVER_notify_transmit_ready (client,
                                          msg_size,
                                          timeout,
                                          &transmit_cb, cls);
// ...
```

Only a single transmission request can be queued per client at the same time using this API. Additional APIs for sending messages to clients can be found in the `gnunet_server_lib.h` header.

5.5 Connecting to CORE

One of the first things any service that extends the P2P protocol typically does is connect to the CORE:

```
struct GNUNET_CORE_Handle *
GNUNET_CORE_connect (struct GNUNET_SCHEDULER_Handle *sched,
                     const struct GNUNET_CONFIGURATION_Handle *cfg,
                     struct GNUNET_TIME_Relative timeout,
                     void *cls,
                     GNUNET_CORE_StartupCallback init,
                     GNUNET_CORE_ConnectEventHandler connects,
                     GNUNET_CORE_DisconnectEventHandler disconnects,
                     GNUNET_CORE_MessageCallback inbound_notify,
                     int inbound_hdr_only,
                     GNUNET_CORE_MessageCallback outbound_notify,
                     int outbound_hdr_only,
                     const struct GNUNET_CORE_MessageHandler *handlers);
```

5.6 Receiving P2P Messages

To receive messages from CORE, services register a set of handlers (parameter `*handlers` in the CORE_connect call) that are called by CORE when a suitable message arrives.

```
static int
callback_function_for_type_one(void *cls,
                               const struct GNUNET_PeerIdentity *peer,
                               const struct GNUNET_MessageHeader *message,
                               const struct GNUNET_TRANSPORT_ATS_Information
                               *atsi)
{
    /* Do stuff */
    return GNUNET_OK; /* or GNUNET_SYSERR to close the connection */
}

/**
 * Functions to handle messages from core
 */
```

```
static struct GNUNET_CORE_MessageHandler core_handlers[] = {
  {&callback_function_for_type_one, GNUNET_MESSAGE_TYPE_MYSERVICE_TYPE_ONE, 0},
  /* more handlers*/
  {NULL, 0, 0}
};
```

5.7 Sending P2P Messages

In response to events (connect, disconnect, inbound messages, timing, etc.) services can then use this API to transmit messages:

```
typedef size_t
(*GNUNET_CONNECTION_TransmitReadyNotify) (void *cls,
                                          size_t size,
                                          void *buf);

struct GNUNET_CORE_TransmitHandle *
GNUNET_CORE_notify_transmit_ready (struct GNUNET_CORE_Handle *handle,
                                   int cork, uint32_t priority,
                                   struct GNUNET_TIME_Relative maxdelay,
                                   const struct GNUNET_PeerIdentity *target,
                                   size_t notify_size,
                                   GNUNET_CONNECTION_TransmitReadyNotify notify,
                                   void *notify_cls);
```

6 Debugging with gnunet-arm

Even if services are managed by `gnunet-arm`, you can start them with `gdb` or `valgrind`. For example, you could add the following lines to your configuration file to start the DHT service in a `gdb` session in a fresh `xterm`:

```
[dht]
PREFIX=xterm -e gdb --args
```

Alternatively, you can stop a service that was started via ARM and run it manually:

```
gnunet-arm -k dht
gdb --args gnunet-service-dht -L DEBUG
valgrind gnunet-service-dht -L DEBUG
```

Assuming other services are well-written, they will automatically re-integrate the restarted service with the peer.

Finally, set the option "DEBUG=YES" to start services with logging of DEBUG messages (if you are using `GNUNET_log` for printf-style debugging).

You should also probably enable the creation of core files, by setting `ulimit`, and echo'ing 1 into `/proc/sys/kernel/core_uses_pid`. Then you can investigate the core dumps with `gdb`, which is often the fastest method to find simple errors.